A Middleware for Exactly-Once Semantics in Request-Response Interactions

Naghmeh Ivaki, Filipe Araujo, Raul Barbosa
CISUC, Dept. of Informatics Engineering, University of Coimbra, Portugal
{naghmeh, filipius, rbarbosa}@dei.uc.pt

***Abstract*—Although the need for the exactly-once request-response interaction pattern is ubiquitous in distributed systems, making it work in practice is anything but simple. Ensuring the at-most-once part of the invocation is relatively easy. Unfortunately, the same is not true for the at-least-once guarantee, which depends on the recovery from crashes of the client, the server and the network. This is what makes the exactly-once interaction so difficult in practice: client and server must log their actions into stable storage, and they must be able to restart the network connections.**

In this paper, we present a middleware that implements the exactly-once request-response pattern, in presence of network and endpoints crashes. The main contribution of our work is to release the programmer from the complex tasks of recovering from message losses and network crashes.

I. Introduction

We no longer need to stress the importance of the Internet for people's life and for businesses. We can find plenty of examples where humans interact with web sites, or programs orchestrate web services to purchase some good or to perform diverse business activities. These interactions often consist of the request-response pattern, where the server performs an action on behalf of the client, sending the result back. Despite being ubiquitous, making this pattern work reliably in the presence of faults is everything but simple. In fact, the client cannot simply invoke the request again, because this may cause the server to repeat *non-idempotent* actions, like making a second reservation for the same person or ordering the same item twice. Other cases have even more stringent restrictions that exceed the *at most-once* semantics, as they actually require an *exactly-once*: employers need to make sure that they issue the paychecks once and only once; the bank must then ensure that it deposits the money once and only once; the same for the corresponding withdrawal operation.

Unfortunately, the most widely used technologies cannot easily ensure exactly-once semantics. Sockets provide the greatest freedom to the programmer, but the programmer must bear the costs of recovering from all failures, like TCP connection crashes. Remote procedure calls may provide some tolerance to faults, or at least hide TCP connection crashes, but their invocation semantics is often at-least-once (RPCs [17]) or at-most-once (RMI [16]). HTTP does not help much either, despite being the protocol used for all web-based requests. Again, programmers must ensure by themselves that HTTP commands provide at-most-once delivery in non-idempotent operations. For this, they usually add an identifier to the request and keep a correspondence between previously seen identifiers and replies on the server side. This is complex, error prone, and repetitive, not to mention all the problems related to the lifetime of the responses in the server. Java Message Service (JMS) is perhaps the simplest technology to provide exactly-once-delivery, due to the notion of transaction available for sending and receiving messages. However, JMS is not tailored for a request-response kind of interaction, because its goal is precisely to decouple parties [6].

Another approach is to use distributed transactions. This ensures that either the client and server agree on the positive outcome of the action, and then, they take it; or they both give up. Unfortunately, this solution has a number of drawbacks: 1) it is difficult to use, as it involves a fairly complex configuration and Application Programming Interface (API); 2) it is heavy, because it involves a coordinator process; and 3) it is slow, due to the several steps involved in protocols like the two-phase commit.

In this paper, we argue that programmers should resort to APIs that provide the exactly-once request-response pattern. Although this kind of pattern occurs repeatedly, writing it correctly is complex, mainly due to the existence of faults that may affect the endpoints or the network. To overcome this problem, we propose and implement a middleware that follows the request/response/acknowledge-response (RRA) protocol of Spector [15]. Our middleware unambiguously defines the boundaries where the client may resend the request or wait, upon recovering from a crash, without taking any chances of repeating the operation. For the transport layer, we enable the user to choose between TCP and a robust socket implementation, named RSocket [5]. Whereas the latter transparently supports network crashes, TCP still forces the user to reconnect, *but* releases the programmer from manually re-synchronizing the state. In both cases, we release the programmer from handling the most complex details of recovering from communication channel failures.

The remainder of the paper is organized as follows: The second section gives a general review of the related work. Section III presents the exactly-once pattern. In Section IV we introduce the middleware that makes this pattern simpler for the application layer. Section V focuses on the recovery form failures. Section VI describes the middleware implementation and API. Next, we evaluate the implementation and conclude the paper.

II. Related Work

"Idempotence" means that performing an operation multiple times will have the same effect as performing it exactly once [9]. Unfortunately, many operations, such as transferring money, or buying a ticket are not idempotent, thus requiring a careful invocation to ensure a single success without repetitions. This section gives a short overview of some techniques that try to achieve this *exactly-once semantics*. This is a strong requirement, because it requires all parties to mask faults. In a request-response interaction, each request must be executed exactly once, and the reply must be delivered to the end user. Most systems only make a best effort at doing this.

In [3], an idempotent messaging protocol is presented that guarantees both message delivery and message idempotence. However, it does not handle the endpoints' failures. Distributed transactions, with their ACID properties (Atomicity, Consistency, Isolation and Durability), are probably the most effective mechanisms to ensure that multiple parties actually agree on the outcome of some interaction. Even in the presence of faults, distributed transactions offer at-most-once semantics: the operation either successfully occurs at all endpoints involved, or nothing occurs and everything reverts back to the initial condition. This is quite close to the definition of the consensus problem [12], where parties must agree on some value from the set they propose. Solving these problems in the general case is impossible [7]. Nevertheless, protocols like the Two-Phase Commit [2] (2PC) can implement distributed transactions in some more restricted, but still quite general, scenarios. 2PC is a heavy and blocking protocol, where all the parties, typically databases, lock the resources involved, until parties take the common final decision. However, in this paper, we are looking for something simpler, because once it operates on the request, the server can make its own state visible despite any further actions from the client. This makes properties like isolation (the 'I' in "ACID") simpler to ensure.

Queued Transaction Processing is another technique to deal with transactional operations [8]. This can also be accomplished with Java Message Service [13]. A client starts a transaction and enqueues the request at the server's queue. Then, the server starts another transaction, dequeues and processes the request, and enqueues the reply at the client's queue. A third transaction is started and the reply is dequeued and processed by the client. Briefly, this technique involves two recoverable queues in front of the server and client, and three distributed commits. Although exactly-once is achieved by making the user a part of the transaction protocol, this model still has its own disadvantages: this approach requires a thick client, which has to be executed on a system that supports transactions; and it involves the cost of three distributed commits. In the case of Java Message Service or some other publish/subscribe technology, a request-response interaction is somewhat contradictory to the spirit of decoupling parties involved in the interaction.

Message Logging is another technique used to ensure exactly-once semantics. This approach is based on logging the state of the interactions and enabling the retransmission of the messages. Phoenix/APP [1] and iSAGA [4] are examples of this technique. Phoenix deals with system failures by logging the interactions and checkpointing the state. The argument is that Phoenix assumes a window of opportunity for the client failure. Exactly-once is achieved only if the client does not fail during this small window. iSAGA saves the system's state in stable storage for every request. This stable storage can be on the client, server or even on another machine. In this system there is no description of the user-to-client interaction. When the client recovers after a crash, there is no guarantees regarding execution semantics, since the recovered state might, or might not, be the latest state presented to the user.

EOS [14] also uses a logging mechanism on the client and server sides, for the web based services to ensure exactly-once service. It handles scenarios where each request may wait for the previous reply. This approach does not deal with connection crashes. It may deadlock when both parties are alive but the TCP connection crashed.

Our exactly-once request-response pattern is quite close to the typical Remote Procedure Calls and Remote Method Invocation paradigms [18], [17], [11], [16], although these provide at-least-once or at-most-once, because their failure handling is much simpler than what we propose in this paper. To extend these approaches, we use a pattern [15] that guarantees the delivery of the messages even in the presence of network and endpoint failures. We use a logging mechanism combined with identified requests and message retransmission to provide exactly-once semantics.

III. The Exactly-Once Request-Response Interaction Pattern

A. Direct Client-Server Pattern

The Request-Response interaction between a client and a server is perhaps the simplest, and yet the most important form of exchanging data we can find in a distributed system. This interaction pattern, occurring in many different applications, operating systems and programming languages, might be used in a number of flavors depending on the semantics of the operation invoked: at-least-once, at-most-once or exactly-once.

We start by putting the operations that the client and server must perform into a sequence. This sequence is based on the work of Spector [15] and assumes the existence of stable storage on the client and server sides, to keep process state in the case of crashes. Despite being important for the practical implementation, the presence or absence of a previous handshake, like a TCP connection, is irrelevant for our discussion at this point. Refer to Figure 1. The client starts by creating a request message, assigning it a unique identifier and storing it with its identifier and timestamp (t_0). The timestamp enables the client to resend the request if it does not receive a reply within some time limit. Next it sends the message to the server (t_1), which receives it (t_2). Since the semantics is exactly-once, we assume that the server cannot generate the same reply again without corrupting its internal state, therefore, the server generates the response and saves it atomically (t_3). This is often simple to do, if the server gets its data from a database, for instance. The application must keep the identifier to avoid repeating the same operation. Then the

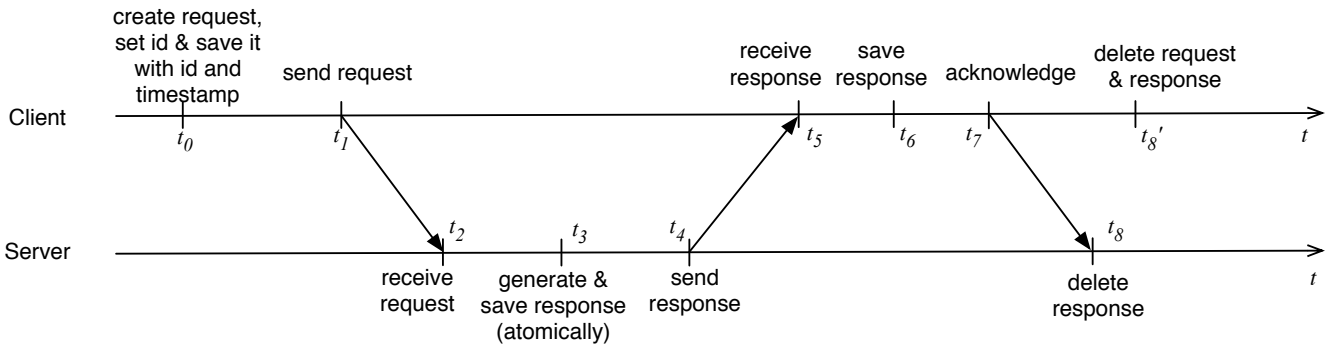

Figure 1. Exactly-Once Request-Response Pattern

server replies (t_4). The client receives the response (t_5), saves it (t_6) and acknowledges (t_7) the server to enable the server to delete the response (t_8). Finally, the client can delete the request and the response ($t_{8'}$). The client can use the response between t_6 and $t_{8'}$.

With this sequence we can ensure the exactly-once semantics, even if the client or the server crash and later recover. Let us examine the client actions. The client creates and stores the request into stable storage, sends it, receives the reply, acknowledges the server and deletes the request from stable storage. We do not care for crashes for time $t < t_0$. For $t > t_{8'}$ the request no longer exists, so we do not care for it either. The client can safely discard any response arriving after that point in time. We thus focus on the client actions in case it crashes for time $t_0 < t < t_{8'}$.

For time $t_0 < t < t_6$, upon recovery, the client cannot determine where it crashed, so it resends the request. After $t > t_6$, the client has a copy of the response, so it no longer needs to resend the request. For $t_0 < t < t_1$, the client never actually sent the request, so it can just send it for the first time when it recovers. If the crash occurs for time $t_1 < t < t_5$, the client will resend the request, but the server can detect the duplicate identifier and avoid re-executing the request. If the client crashes in the interval $t_5 < t < t_6$, upon recovery, it will resend the request as it does not have the corresponding response in its stable storage. In this case, it is up to the server to filter duplicate requests and resend the response. For time $t_6 < t < t_7$, the client just picks the response it saved. For time $t_7 < t < t_{8'}$, the client will resend an acknowledgement. The server could then receive an acknowledgment for a response it no longer has, but it can safely discard the acknowledgment.

Now, let us do a similar analysis for the server. If the server crashes before $t = t_3$, upon recovery it cannot do anything, because it no longer has the request. This client must resend the request after a given timeout in this case. For time $t_3 < t < t_4$, the client will resend the request since it has not received the response yet. In this case, the server must retrieve the reply from storage instead of re-executing it. For time $t_4 < t < t_8$, the server cannot know whether or not the reply reached the client. It must wait for either the acknowledgment from the client or for a repetition of the request.

B. Demonstration of Correctness

In this section we formally identify the properties of the exactly-once request-response pattern and demonstrate that they hold. We assume that the channel does not create, change, or reorder (First In First Out – FIFO) any messages. We assume that channels, client, and server eventually will be correct for a sufficiently long time that enables their interaction to finish. Regarding safety and liveness, we require the following properties for the exactly-once request-response pattern:

Safety 1 At-most-once execution of requests.
Safety 2 No invention of response.
Safety 3 No duplication of response.
Liveness 1 At-least-once reception of response.

Liveness says that all requests eventually have a response. This is the "at-least-once" part of the interaction. However, beside a live behavior, we must ensure some safety properties, to prevent double execution of the request (Safety 1) or reception of the message (Safety 3), and to prevent receiving a response for a request the client never did (Safety 2). Next, we demonstrate these properties.

a) Safety 1: At-most-once execution of a request: One request is univocally identified by a number. Therefore, if the server checks its stable storage for this request's identifier, it may not execute the same request twice between t_3 and t_8. We must demonstrate that after t_8, the server may not execute the same request again. Once the client sends the acknowledgment (t_7), it will not send the same request again, because it saved the response in the stable storage at time $t = t_6$. Since we rely on FIFO channels, the server must not receive any repetition for the same request after $t = t_8$.

b) Safety 2: No invention of response: This property derives from the fact the channel does not invent any messages.

c) Safety 3: No duplication of response: This property depends on the implementation of the client, which must use the response during the interval $t_6 < t < t_{8'}$ and must atomically delete the response and finish the task it has to do with the response at time $t = t_{8'}$. In this case, if the client ever crashes for $t < t_6$, when it restarts it did not use the response. For time $t > t_{8'}$, it is no longer waiting for the response, so it will discard it (it may receive two or more responses in fact). For time $t_6 < t < t_{8'}$, the effect of the response takes place only once.

d) Liveness 1: At-least-once reception of response: First, we assume that the client periodically resends the request if it does not get any response, e.g., because the channels keep failing. Assuming these conditions, we need to prove that the client reaches the point where it saves the response (t_6). Since the client keeps resending the request, and the server always gives a response to this request (either by executing the action or getting the response from stable storage), the property follows from the assumption of correctness of the channels, client and server for a sufficiently long time.

IV. The Middleware Layer

A. Description

Apart from generating the request and the response, most actions that clients and servers perform are repetitive and might be handed over to library functions. In Figure 2, we show the result of our approach to factor out client and server actions from Figure 1. We separate the endpoint applications in two layers: the application layer takes care of generating and using the messages, while the middleware layer is partially responsible for message idempotence and guarantees the delivery of the requests and responses to the application layer.

With middleware, the client no longer resends requests. Therefore, it does not need to generate identifier or timestamps for the messages. This is the most important difference for the client. Once the client returns from sending the request, in action t_1, it does not need to keep re-sending the request, although the middleware allows this, if the client crashes and later recovers. The server side has an important simplification as well: after delivering the response to the middleware, the application layer may delete the response. This allows the server to use a typical API with a single blocking point to receive requests, discarding the need for a second one to receive the acknowledgments. We describe the details in the following paragraphs.

The client starts by creating a request message (t_0). The underlying middleware generates a unique identifier for the message, from its contents (its hash or even the message itself could serve as identifier). If the client resends the same message, the middleware can determine if it is still processing that message or if it is new, and, at the same time, releases the application from the burden of managing message identifiers[1]. The middleware saves the message, together with the identifier and a timestamp (t_1). Next, the client-side middleware sends the message to the server middleware (t_2), which receives and delivers it to the server application (t_3). The server middleware must not send a new request for the same identifier before the application replies (something that could happen if the client sends the same request twice or more). Up to $t = t_5$ this goes on unchanged. However, to enable the server application layer to delete all data related to the request, the server middleware must save the response at $t = t_{5'}$. The server middleware replies to the client middleware (t_6), which receives (t_7) and saves the response (t_8). This will enable the server side to delete the response ($t_{11'}$), once the client middleware acknowledges the reply ($t_{9'}$), before or after the middleware delivers the response to the client. The remaining actions of the client are the same as in Figure 1.

If the client crashes between $t_0 < t < t_{10}$, it will resend the request after resuming. After that point, either it still has the response or it no longer has the request (after $t > t_{12}$), so it no further interacts with the middleware. In the interval $t_1 < t < t_8$, the client middleware already saved the request, so it can resend the request by itself. However, the application layer may also resend the request. This is not a problem because the identifier will be the same and the middleware is able to filter the duplication. If the crash happens in the interval $t_8 < t < t_{10}$, the client middleware still has the response and does not need to resend the request to the server. On the server side, if a repetition of the request arrives during the interval $t_5 < t < t_{11'}$, the middleware itself will provide the reply. Before that point, the server application layer itself must keep the response and corresponding identifier to filter duplicates.

B. Exactly-Once Properties

a) Safety 1 (At-most once execution of a request): Assume that a repeated request arrives after $t > t_{5'}$ at the server application layer. Then, since the server middleware stores the response until $t = t_{11''}$, the client middleware must have resent the request after acknowledging it, because the channel is FIFO. However, since the client middleware saved the request at time $t = t_8$ and deletes it at $t = t_{11'}$, the originator of the repetition must be the client application layer instead. But this is contradictory to the fact that after $t = t_9$ the client cannot have repeated the request, because it already saved the response, before deleting everything at $t = t_{12}$.

b) Safety 2 & 3 (No invention of response and no duplication of response): These are very similar to the case of Figure 1, which we demonstrated before.

c) Liveness 1 (At-least-once reception of response): We used the Spin model checker [10] to formally verify the liveness claims of our approach. Model checking is a method for verifying whether a specification is fulfilled by a model. A specification is a set of properties which a system is expected to satisfy, and a model is a formal description of the system's behavior, written in a modeling language, intended to preserve as much detail as necessary for the verification. Model checking tools such as Spin take a model and its specification as input. Their output is either an indication that the model is correct, or a case in which the correctness properties fail to hold.

The Spin model checker accepts a formal modeling language called Promela. This language is appropriate to model distributed software systems. Inter-process communication can be specified using message channels, which can be either synchronous or asynchronous. We used asynchronous channels to model the communication between the client middleware and the server middleware. Synchronous channels were used to model the call-return interactions between the client application and the client middleware, as well as between the server application and the server middleware.

[1] A shortcoming of this approach is that the client cannot generate a new request with the same data before acknowledging the previous one. Nevertheless, this limitation can be trivially solved by adding some salt to change the message.

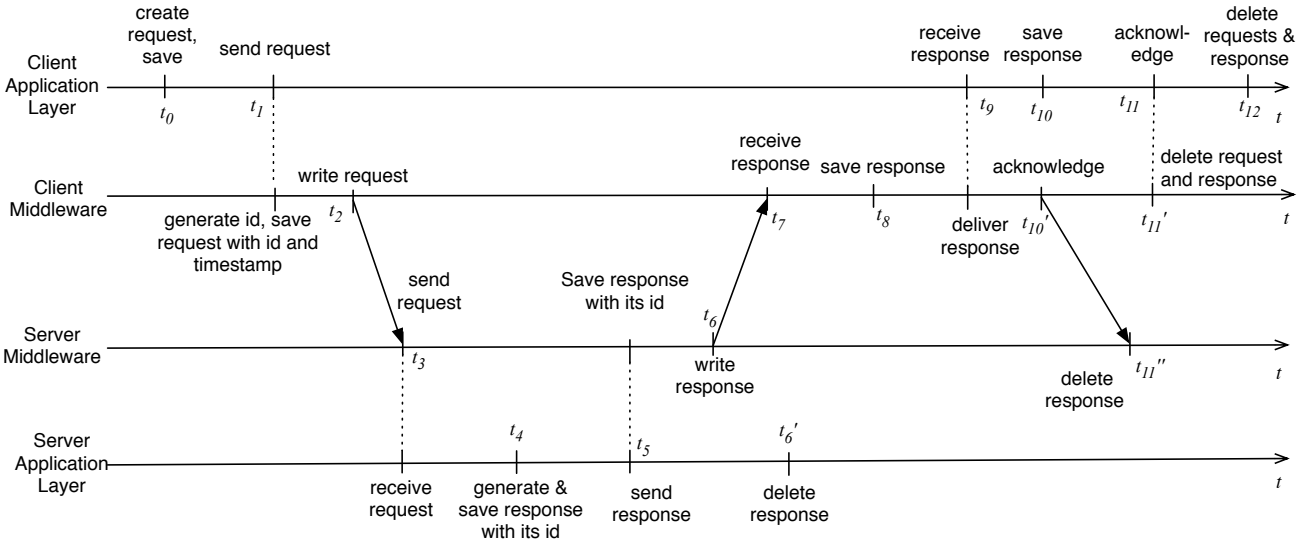

Figure 2. Exactly-Once Request-Response Pattern With Middleware Layer

We modeled a system consisting of four processes, namely the client- and server-side applications and the corresponding middleware instances. An additional process was constructed in order to model failures. Two kinds of failure are considered: lost messages between the two middleware processes, and crashes of either client or server applications. Lost messages were modeled by snatching messages from the channels between client and server middleware. Crashes of the client and the server were modeled by resetting the client- or the server-side (including the application and the middleware) to the initial state. Only the content of the stable storage is assumed to be preserved after a crash.

The interaction among client application, client middleware, server middleware, and server application was modeled according to the specification of our approach. The model describes the behavior of the system for a single request. Given that multiple requests have no influence on one another and that all tiers are able to distinguish between different requests, this abstraction allows us to reduce the state-space needed to model the system.

Our analysis using model checking focused on the system's liveness. Given that the communication channel between the client and the server is asynchronous and that any messages may be lost, liveness can only be guaranteed under the assumption that eventually there is a fault-free period of execution. Moreover, as processes may crash, one must assume that these will also eventually remain fault-free for some period of the execution. To model this assumption, we allow the failure-injecting process to terminate its execution, and specify that a response is eventually received by the client if there are no more failures. In linear temporal logic (LTL) the specified property is the following:

$$\Box(\text{faultfree} \rightarrow \Diamond\text{terminates})$$

The formula should be interpreted as: whenever the system becomes fault-free (i.e., the failure injector ends its execution) the client will eventually receive the response and therefore terminate the execution of a request. The symbol "faultfree" was defined as the failure injector ending its execution and the symbol "terminates" was defined as the client application reaching its final statement.

The correctness of the model with respect to the liveness claim was checked using Spin version 6.1.0. The formula is found to be correct by the verifier in 2 hundredths of a second, for a system totaling 5.8×10^3 states and 2.2×10^4 transitions (with partial order reduction enabled). This increases our confidence in the correctness of our approach regarding liveness.

V. Recovery From Crashes

Difficulties in ensuring the exactly-once semantics come from the unreliable and faulty nature of the endpoints and network environment. As we demonstrated before, if processes have access to stable storage, they can recover from failures and use our middleware to ensure the exactly-once semantics. In this section, we go through the recovery process in the connection, client and server.

Reliable Channel: As we mentioned before, the channel must provide some guarantees to the exactly-once pattern, namely FIFO delivery, no invention and no change of messages. The TCP protocol provides all of these guarantees. However, TCP cannot provide a liveness guarantee, because connections do not recover from crashes. This leaves the burden of rolling back applications to a coherent state to the programmers. In our approach, when a plain TCP connection fails, the middleware returns an exception to the application. Since the client-side middleware relies on a timeout to resend requests that lack a response, it imposes no special request on the application layer for re-synchronizing client and server. After a TCP crash, the programmer just needs to provide a new connection between the same endpoints. This, we believe, is one of our most interesting contributions, because this is a very error-prone task.

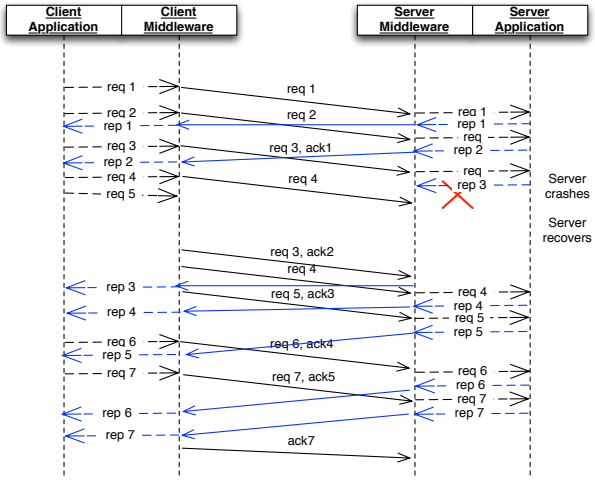

Figure 3. Recovery From Server Crash

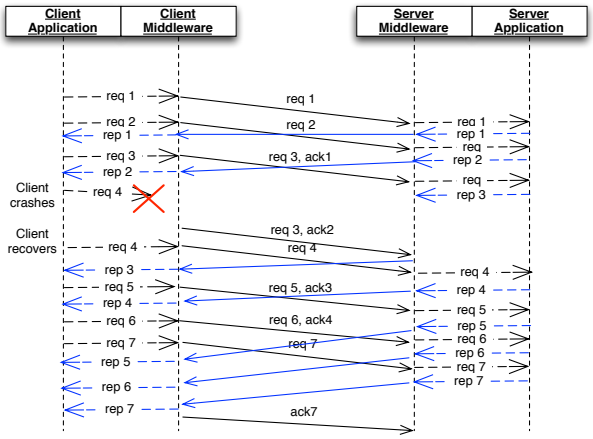

Figure 4. Recovery From Client Failure

We go one step further: to address the reconnection problem entirely, we use a technology that abstracts away the communication channel problems: Robust Socket or RSocket [5]. It offers end-to-end reliability, by hiding TCP connection crashes from the application layer. It transparently takes care of buffering and resending messages after recovery from failures. We thus have two options: either the application takes care of reconnecting by itself, *or* it hands everything to the Robust Socket implementation. As we shall show experimentally, there is a tradeoff involved due to the lower overheads of TCP and possibly greater control offered to the programmer.

Server Failure: When a server crashes, the client-side middleware must take the initiative of recovering a connection and resending the requests. The client keeps trying to reconnect periodically until the server recovers, either explicitly if using TCP or automatically with RSocket. After recovery, the server looks for failed connections in its stable storage and provides information of those connections to the middleware.

Refer to Figure 3. The state of the communication before the server crashes is the following: requests 1 and 2 (req1 and req2 in the figure) are sent and replied and their replies are delivered to the application layer; req3 is processed by the server application and is given to the middleware for sending but the server crashed before sending the reply to the client; req4 and req5 are sent to the server but the server crashed before receiving the requests. Once interaction is resumed, the client middleware resends req3, req4, and req5 since they were not replied within the expected time. Then, the server middleware sends the reply of req3 from its recovered state and delivers req4 and req5 to the server application for processing. The server treats the remaining requests in a standard way.

Client Failure: When the client restarts after a crash, it checks its log files for failed connections. Once it recovers the state of the failed connections, it asks the client middleware to recover the failed connection's state from its stable storage. The client middleware and the client application resend the requests with pending replies. Figure 4 demonstrates a simple scenario with the client failure and recovery. The last state of the interaction, before the client failure, is the following: req1, req2, and req3 are sent to the server; req4 is given to the client middleware but the client crashed before sending the request to the server; the server already processed req1, req2 and req3; rep1 and rep2 are delivered to the client application, but rep3 could not be sent because the client crashed. The client recovers after a while and resends the pending requests. The req3 and req4 are sent by the client middleware. The client application may also resend req4, because it is not replied yet, but the middleware will ignore it, since it is duplicated. On the server side, the server middleware sends rep3 and delivers the req4 to the server application for processing. The client and server continue interacting normally.

VI. Implementation

A. Main Options

We used Java to implement the exactly-once middleware pattern as a library. We tried to keep several potential sources of overhead under control: 1) the load on the network caused by messages and their headers; 2) memory footprint of the middleware, including message buffers; 3) the load imposed by the logging operations of the applications. To reduce the overhead of middleware messaging, and memory usage, we used some straightforward techniques, like piggybacking the acknowledgements in the requests, and immediately deleting unnecessary data from buffers. To reduce the load imposed by the middleware logging system, we used the following technique. A timer is scheduled to periodically log the complete state of the middleware (with all information regarding the messages). As the middleware state may change during two consecutive logging times, the middleware keeps any occurring change (e.g., removing a message, changing the timestamp of the message when it is retransmitted) in a distinct file. In case the middleware needs to recover the last state, after some failure, it will retrieve the whole (periodic) state of the middleware, plus the partial updates that occurred afterwards. The logger deletes all these partial updates, as well as the previous periodic version, whenever it writes a new one.

Table I
MIDDLEWARE API

Middleware API
`ClientMiddleware(), ServerMiddleware()`
`addConnection(socket)`
`removeConnection(socket)`
`MsgProperties read(socket, message)`
`int write(socket, message)`
`acknowledge(socket, messageID)`
`recover()`
Logger API
`logRequest(socket, inputdata)`
`deleteRequest(socket, inputdata)`
`logResponse(socket, responseID, outdata)`
`deleteResponse(socket, responseID)`

B. The Exactly-Once Middleware API

In Table I, we show the main methods of the Middleware and Logger API. We remove class references whenever possible to simplify presentation. The Middleware API provides the following operations to the applications: creating the middleware, adding and deleting a connection to/from the middleware, writing and reading to/from the middleware and acknowledging (for the client only). Creation of a middleware object is asymmetric for the two peers. The server creates a `ServerMiddleware` object, whereas the client creates the middleware by instantiating a new `ClientMiddleware` object. Despite using different classes, client and server use symmetric operations most of the time.

Once initialized, both client and server can delegate any new or existing connections to the middleware using the `addConnection` method. TCP, and RSocket are supported in the middleware. Afterwards, the client and server can interact, using `read`, `write` and `acknowledge` methods. Since the middleware might be handling several connections simultaneously, the application must specify the socket they are using for each operation. Once terminated, the application can remove the connection using `removeConnection`.

The `write` operation either returns the identifier of the message to the application or -1 to inform the application of a message duplication. With the `read` operation, an object `MsgProperties` is returned to the application as well as the message itself. This object encapsulates the properties of the message, including the message identifier and message size. A connection can be removed from the middleware layer using the `remove` operation. To recover a connection after a crash, we must use the `recover` method. For simplicity we do not include its parameters here.

We also provide a `Logger` class with the facilities we described in Section VI-A. This logger class offers methods to log the requests, the responses, and to delete them. The idea is to release the application developer from the need to efficiently implement these functionalities. With the help of the `Logger` class, the application can go through the steps of Figure 2, although some servers may rely on mechanisms like database transactions, to keep their state coherent.

VII. EVALUATION OF THE MIDDLEWARE

A. Experimental Setup

To examine the performance and for comparative purposes, we examined the latency and throughput, with and without middleware for TCP and RSocket. We also compared our middleware with Java RMI, because it is a popular technology, which offers at-most-once semantics. To test the core functionalities of the middleware we emulated connection, client and server failures. In all the experiments, we monitored memory and CPU usage.

The operations provided by the servers are named “Echo”, “Invoke” and “Invoke2”. The Echo operation receives a String object and returns the object back to the client without any changes. Invoke receives an Empty object, reads a name and an age from a file, and sends a new object with a name and an age back to the client. Invoke2 receives an Exchange object, writes its name and age into a file, reads a new name and age from a file, makes a new object with the new name and the new age, and sends it back to the client. We performed a baseline evaluation to understand the typical execution time of these operations and observed that, on average and for a total of 100 repetitions, Echo takes 0.13 ms to execute, Invoke takes 0.34 ms, and Invoke2 0.55 ms.

To examine the latency, we send a request to the server and calculate the time taken to receive the reply from the server. To examine the throughput, we use 1000 requests and consider the time it takes until the client gets the last reply. To minimize environmental effects on the experiments and possible warm-up periods, we repeated the tests 100 times and ignored the first 10 results for latency and the first 20 for throughput. To evaluate the middleware core functionality, we designed a set of tests that emulate the connection, client and server failure. We use Oracle’s JConsole, to monitor memory and CPU usage of the client and server. The client sends a number of requests to the server with the rate of 100 messages per second, taking almost 5 minutes in a fault free execution. The failure occurs at minute 2 and takes one minute. After recovery from the failures, the client and server continue communication normally.

We ran the client on a Mac OS X, version 10.6.7, with a 2.4GHz Intel Core 2 Duo processor, 4GB of RAM and 3MB of cache. The server ran Linux version 2.6.34.8, with a 2.8 GHz Intel processor, 12 GB of of RAM and 8 MB of cache. Both client and server were in the same local network.

B. Fault-Free Experiments

The goal of the fault-free experiments is to set the performance figures under normal circumstances. We used one client and 10 simultaneous clients for the purpose. We show all the results in Table II.

Results show that plain TCP is usually the fastest protocol, except (by little) for the Echo operation with one client, the fastest operation. Although TCP’s throughput with one client is somewhat higher than RMI’s, it is not much higher with ten clients. For the Invoke2 operation it is even worse than RMI. We believe that this is caused by the overhead of the multiple threads handling the TCP connections on the server

Table II
LATENCY AND THROUGHPUT WITH AND WITHOUT MIDDLEWARE

1 Client						
Latency (ms)						
	Plain			With Middleware		
	Echo	Invoke	Invoke2	Echo	Invoke	Invoke2
TCP	1.14	1.54	2.09	2.58	2.94	3.51
Rsocket	42.12	42.98	43.09	43.88	44.11	44.26
RMI	1.07	1.59	2.36			
Throughput (requests/s)						
	Plain			With Middleware		
	Echo	Invoke	Invoke2	Echo	Invoke	Invoke2
TCP	41719.84	14425.05	2021.97	7716.65	6436.45	1662.61
Rsocket	35329.49	11694.72	2003.76	7390.27	6374.60	1482.38
RMI	1318.81	1159.89	671.08			
10 Clients						
Latency (ms)						
	Plain			With Middleware		
	Echo	Invoke	Invoke2	Echo	Invoke	Invoke2
TCP	2.05	2.85	3.54	8.44	8.45	8.83
Rsocket	42.69	43.15	43.20	43.97	44.71	45.11
RMI	2.69	3.48	4.17			
Throughput (requests/s)						
	Plain			With Middleware		
	Echo	Invoke	Invoke2	Echo	Invoke	Invoke2
TCP	3546.73	1907.93	151.06	672.50	404.00	113.38
Rsocket	3377.56	1772.83	145.25	612.54	390.38	118.54
RMI	1031.16	974.36	174.77			

side: we create one thread for each new connection, whereas RMI uses a thread pool. Creating a new thread would be more appropriate if the operations are relatively long, but for short operations, a thread pool is more efficient.

Since RSocket is implemented over TCP, we were expecting worse performance from this technology. In fact, we observed that latency in RSocket is unreasonably high, unlike throughput, which is relatively close to that of TCP. We discovered that the reason for the poor latency comes from Nagle's algorithm, which is impossible to switch off in RSocket.

To better observe the degradation of performance caused by the middleware, in Figure 5 we show latency, throughput, and the degradation (in percentage) caused by the middleware. The middleware causes a sensible overhead on latency and throughput, compared to plain TCP. We could say that this is expected, due to the logging operations of the exactly-once semantics. On the other hand, since RSocket has a high latency, the overhead caused by the middleware is insignificant (Figure 5 (1) and (2)). We cannot achieve the same results for throughput. Although our results show that the middleware has some overhead on the latency and throughput, we believe that this overhead may be negligible in many real applications. Whenever the processing time of the server increases, as with the Invoke2 operation, latency and throughput degradation soften, thus becoming less important.

In Figure 6 we show results in another way, including the comparison between the middleware and RMI. We show the standard deviation as bars in the respective columns. Although RMI shows worse latency than plain TCP in most settings (refer to Table II), it has better latency than the middleware with both plain TCP and RSocket (refer to Figure 6 (1) and (2)). The interesting thing is that our middleware handled more requests per second from a single client, but not with ten clients (Figures 6 (3) and (4)). This shows that the middleware does not get in the way of this particular item of performance as it does for latency, but again, we have the effect of the server thread pool pushing for RMI with 10 clients.

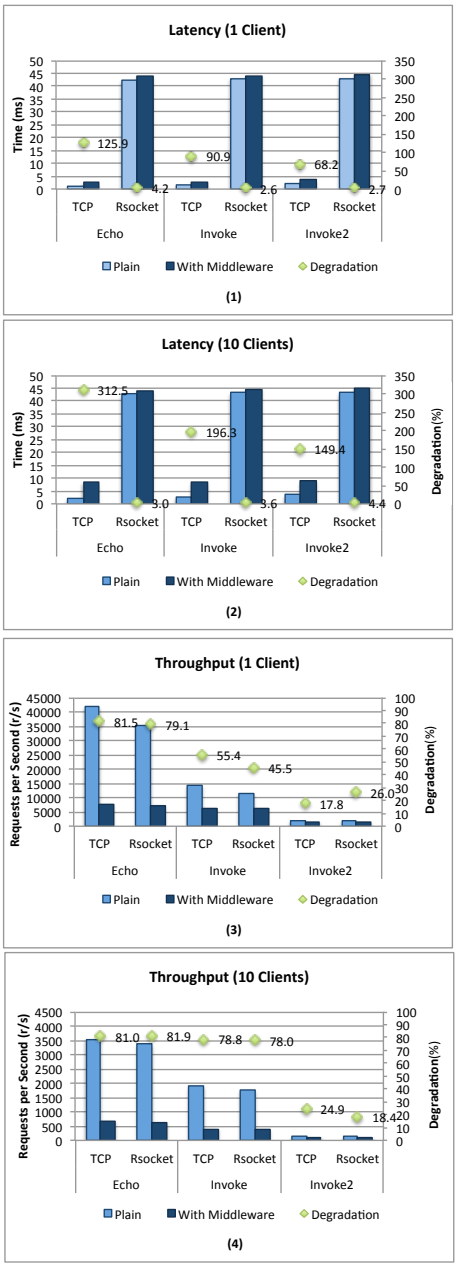

Figure 5. Latency and Throughput Degradation Caused by Middleware

C. Failure and Recovery

In this section we evaluate the middleware and RSocket in an emulated faulty environment. We defined three scenarios: connection failure, client failure, and server failure. We force these failures to the system at minute 2 until minute 3, and monitor the memory and CPU usage of the client and server during the tests.

Refer to Figure 7. The four columns are the client memory usage, the client CPU usage, the server memory usage, and the server CPU usage. The memory usage plots include instantaneous values as well as a more stable progressive average, which includes all past values[2]. Each of the five rows of the figure displays the results of a different test:

[2]The n-th instance is $\left(\sum_{i=1}^{n} d_i\right)/n$, where d_i is the i-th instance of data.

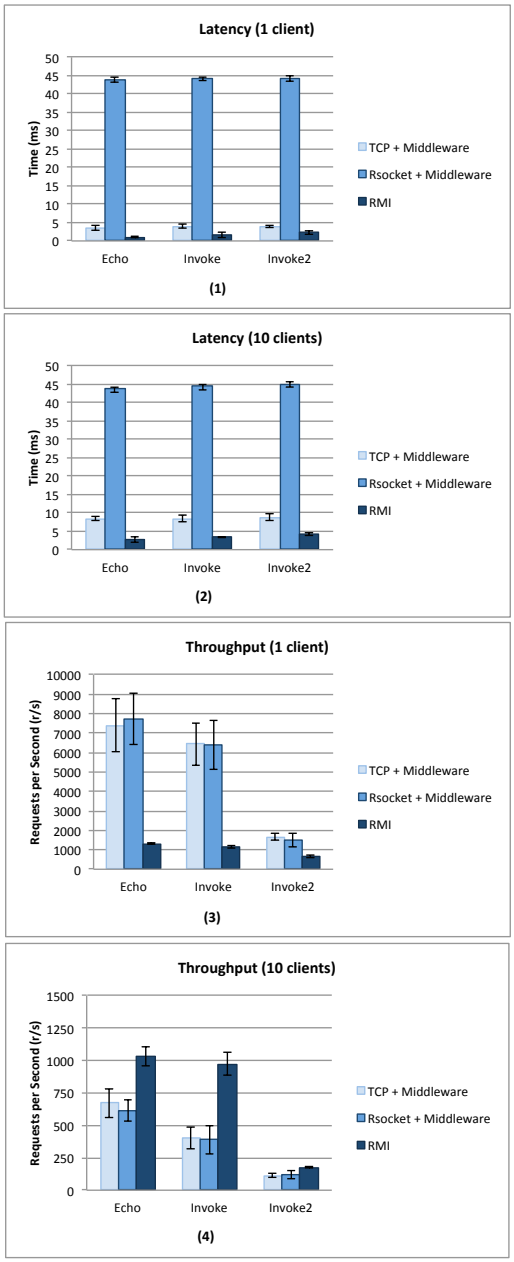

Figure 6. Latency and Throughput Comparison with RMI

with raw RSocket; with middleware and RSocket in a fault free run, with connection failure, with client failure, and with server failure. Comparing the first two rows, plain RSocket and RSocket plus middleware, we observe that the middleware's overhead on the memory is 7.21% (about 0.76 Mb extra memory) and on CPU usage is 66.10% (about 1.95% extra CPU usage). On the server side, the overhead on the memory is 44.11% (about 0.45 Mb extra memory) and on the CPU is 18.87% (about 0.60% extra CPU). In the third row of the figure we show the results of a connection failure. When a connection fails, both client and server stop doing processing related to that connection. Client's RSocket keeps trying to make a new connection. Client's CPU usage, during the failure, shows this fact as well. Since RSocket uses exponential back-off strategy to increase the delay between two consecutive reconnection attempts, reconnection and recovery does not occur exactly at minute 3. Client heap memory starts filling smoothly until the new connection is created. On the server side, the figure does not show any clearly visible event related to the memory usage, although CPU usage decreases until the new connection is created. The results with client failure are shown in the fourth row of the figure. The client memory usage drops to zero after the crash and fills up when the client recovers. After the client restarts, it reads the log files and recovers the state of the failed connection. As the figure shows CPU usage shows a peak for starting and recovering. While the normal starting time for the client was 55 ms, it took 85 ms to start and recover the connection's state. The next peak in the client side is caused by the middleware logging system. Server does not show any sensible effect on the memory during the client failure, but the CPU usage decreases until the new connection comes. The fifth row in the figure shows the server failure scenario. During the failure, the client periodically keeps trying to reconnect, as RSocket cannot handle this. We defined 5 seconds for the delay between two consecutive attempts. When the server recovers after the failure, it reads the log files to recover the state of the failed connections. It takes a large CPU usage as the figure shows. While the normal starting time for the server was 23 ms, it took 37 ms to recover from the failure. Client also uses more CPU to reconnect to the server and resend the requests pending for the reply.

VIII. Conclusions and Future Directions

In this paper we presented an exactly-once middleware that releases the programmer from most of the complications deriving from network crashes (including those caused by end-point crashes). Our middleware supports two types of sockets: TCP Java Socket or Robust Socket [5] (RSocket). These solutions represent a compromise: the RSocket implementation transparently recovers from network crashes, but offers worse performance, in part due to an incomplete implementation. On the contrary, the TCP solution does not recover transparently and forces the programmer to manually reconnect. However, even in this case, we take care of re-synchronizing the state between client and server, thus releasing the programmer from most of the burden of this operation. Furthermore, the standard TCP sockets still enable all the normal options, such as deactivating Nagle's algorithm, something that is currently not possible with the RSocket.

Our experiments show the feasibility of our approach. For example, performance results with TCP are in line with and sometimes better than with RMI, a standard, well-matured technology that offers at-most-once semantics. We also showed that our middleware has a small impact in terms of memory and CPU costs. Recovery time from crashes mainly depends on the policy adopted by the underlying sockets. In the case of Robust Sockets, this is an exponential back-off, but a TCP implementation might adopt some other more active (thus costly) approach.

For these reasons, we argue that our exactly-once pattern provides clear advantages to programmers: it departs as little as possible from the ubiquitous and very simple at-most-once

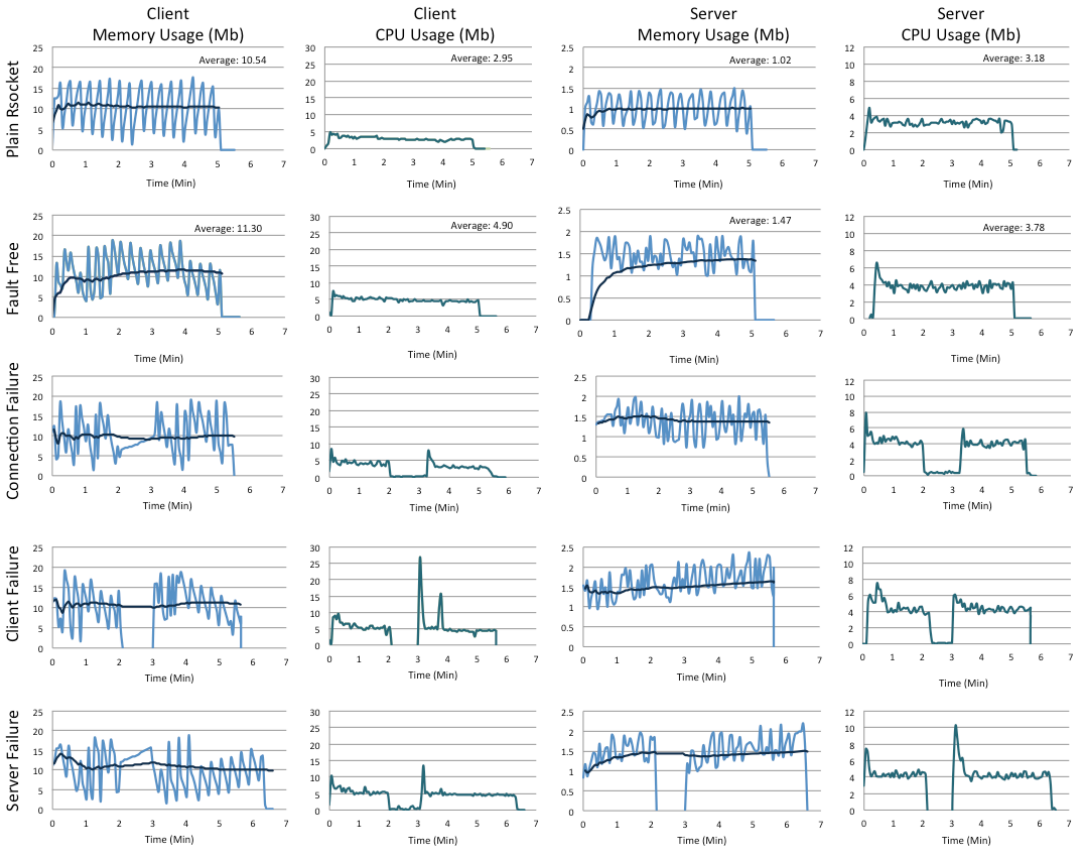

Figure 7. Memory and CPU usage with Middleware and RSocket under Connection, Client and Server Failure

request-response, it turns connection crash recovery nearly or completely transparent, and it does not compromise performance. An interesting future direction would be to make the pattern independent from a specific programming language and make it available system-wide.

Acknowledgments

This work was partially supported by the the Portuguese Foundation for Science and Technology contract SFRH/BD/67131/2009 and by the project CMU-PT/RNQ/0015/2009, TRONE — Trustworthy and Resilient Operations in a Network Environment.

References

[1] R. Barga, D. Lomet, S. Paparizos, Haifeng Yu, and S. Chandrasekaran. Persistent applications via automatic recovery. In *Database Engineering and Applications Symposium, 2003. Proceedings. Seventh International*, pages 258–267, July 2003.

[2] B.S. Boutros and B.C. Desai. A two-phase commit protocol and its performance. In *Database and Expert Systems Applications, 1996. Proceedings., Seventh International Workshop on*, pages 100 –105, September 1996.

[3] Jeremy Brown, J. P. Grossman, and Tom Knight. A lightweight idempotent messaging protocol for faulty networks. In *Proceedings of the fourteenth annual ACM symposium on Parallel algorithms and architectures*, SPAA '02, pages 248–257, New York, NY, USA, 2002. ACM.

[4] Kaushik Dutta, Debra E. VanderMeer, Anindya Datta, and Krithi Ramamritham. User action recovery in internet sagas (isagas). In *Proceedings of the Second International Workshop on Technologies for E-Services*, TES '01, pages 132–146, London, UK, UK, 2001. Springer-Verlag.

[5] Richard Ekwall, Péter Urbán, and André Schiper. Robust TCP connections for fault tolerant computing. In *In Proc. 9th International Conference on Parallel and Distributed Systems (ICPADS*, pages 501–508, 2002.

[6] Patrick Th. Eugster, Pascal A. Felber, Rachid Guerraoui, and Anne-Marie Kermarrec. The many faces of publish/subscribe. *ACM Comput. Surv.*, 35(2):114–131, June 2003.

[7] Michael J. Fischer, Nancy A. Lynch, and Michael S. Paterson. Impossibility of distributed consensus with one faulty process. *J. ACM*, 32(2):374–382, April 1985.

[8] Jim Gray and Andreas Reuter. *Transaction Processing: Concepts and Techniques*. Morgan Kaufmann, 1993.

[9] Pat Helland. Idempotence is not a medical condition. *Queue*, 10(4), 2012.

[10] Gerard J. Holzmann. *The SPIN Model Checker: Primer and Reference Manual*. Addison-Wesley, 2003.

[11] Rüdiger Kapitza, Michael Kirstein, Holger Schmidt, and Franz J. Hauck. *FORMI: An RMI Extension for Adaptive Applications*. 2005.

[12] Leslie Lamport. Paxos Made Simple. *SIGACT News*, 32(4):51–58, December 2001.

[13] Richard Monson-Haefel and David Chappell. *Java Message Service*. O'Reilly & Associates, Inc., Sebastopol, CA, USA, 2000.

[14] German Shegalov, Gerhard Weikum, Roger Barga, and David Lomet. EOS: Exactly-Once E-Service middleware. In *In Proc. of the 28th Conference on Very Large Databases*, pages 1043–1046. VLDB Endowment, Morgan Kaufmann, 2002.

[15] Alfred Z. Spector. Performing remote operations efficiently on a local computer network. *Commun. ACM*, 25(4):246–260, April 1982.

[16] Sun Developer Network. RMI remote method invocation. http://www.oracle.com/technetwork/java/javase/tech/index-jsp-136424.html. Page visited on June 2, 2012.

[17] Inc. Sun Microsystems. RPC: Remote Procedure Call Protocol Specification Version 2. RFC 1057, June 1988. Obsoletes RFC 1050.

[18] B. H. Tay and A. L. Ananda. A survey of remote procedure calls. *SIGOPS Oper. Syst. Rev.*, 24(3):68–79, July 1990.